RESYN: A Generalized Recursive Regular Expression Synthesis Framework

Seongmin Kim[1] , **Hyunjoon Cheon**[2] , **Su-Hyeon Kim**[3] , **Yo-Sub Han**[3] and **Sang-Ki Ko**[1]
[1]University of Seoul
[2]Gyeongsang National University
[3]Yonsei University
{mrseongminkim, sangkiko}@uos.ac.kr, hyunjooncheon@gnu.ac.kr, {suhyeon.kim, emmous}@yonsei.ac.kr

Abstract

Existing Programming-By-Example (PBE) systems often rely on simplified benchmarks that fail to capture the high structural complexity of real-world regexes, such as deeper nesting and frequent use of union operations. To overcome the resulting performance drop, we propose RESYN, a synthesizer-agnostic divide-and-conquer framework that decomposes complex synthesis problem into manageable sub-problems. We also introduce SET2REGEX, a parameter-efficient synthesizer capturing the permutation invariance of examples. Experimental results demonstrate that RESYN significantly boosts accuracy across various synthesizers, and its combination with SET2REGEX establishes a new state-of-the-art on challenging real-world benchmark. The complete source code, datasets, and pre-trained model checkpoints are publicly available at https://github.com/mrseongminkim/ReSyn.

1 Introduction

Regular expressions (regexes) are a fundamental tool in a wide range of domains, including text processing, data validation, and network security. Despite their expressive power, regexes are notoriously difficult to write correctly due to their complex and unintuitive syntax, even for experienced practitioners. To mitigate this difficulty, Programming-by-Example (PBE) approaches, which automatically synthesize regexes from input-output examples, have been studied extensively.

Recently, deep learning-based synthesizers leveraging large language models (LLMs) and encoder-decoder architectures have demonstrated promising results, achieving higher accuracy and significantly faster synthesis compared to traditional enumeration-based methods [Vaduguru *et al.*, 2024]. However, we argue that existing neural approaches suffer from a fundamental limitation: they are evaluated under overly simplified settings that fail to reflect the true structural complexity of real-world regexes.

Our analysis reveals a critical disconnect between existing benchmarks and practical regexes. Prior works often rely on synthetic, domain-specific, or simplified datasets that restrict character classes and operators, artificially lowering synthesis difficulty. As summarized in Table 1, genuine regexes from *RegExLib* are structurally deeper and semantically richer, featuring over $2\times$ more abstract syntax tree (AST) nodes than simplified versions and $3.6\times$ more than synthetic benchmarks. Furthermore, while existing benchmarks are predominantly linear, real-world regexes heavily utilize the Union operator. This discrepancy suggests that state-of-the-art models may overfit to simplified structures, failing to generalize to the recursive and nested nature of true practical instances.

Dataset	Category	Depth	Nodes	Alt
[Vaduguru *et al.*, 2024]	Synthetic	3.39	6.45	0.04
[Ferreira *et al.*, 2021]	Domain	3.03	8.02	0.05
Snort[1]	Simplified	3.05	8.11	0.05
	Real-World	3.13	9.60	0.06
RegExLib[1]	Simplified	3.31	11.04	0.48
	Real-World	**4.40**	**23.69**	**1.25**

Table 1: Comparison of structural complexity. To ensure consistency, all regexes were standardized using our Canonicalizer (§3). Statistics are based on unique AST structures, abstracting concrete literals. Simplified datasets are from SPLITREGEX [Kim *et al.*, 2025]. Alt denotes the average number of Union operators.

This structural complexity poses severe challenges to existing paradigms. Enumeration-based solvers scale poorly with regex depth, while neural sequence-to-sequence (Seq2Seq) models suffer from two architectural inefficiencies: they flatten hierarchical ASTs into linear sequences, losing long-range dependencies, and they treat input examples as ordered sequences, violating the permutation invariance of sets. Although some divide-and-conquer strategies have been explored, they typically rely on rigid heuristics, such as assuming top-level Concatenation, and lack the true recursive capabilities needed for complex scenarios.

To overcome these limitations, we propose RESYN, a three-stage framework spanning data, model, and algorithm. We first introduce a Regex Canonicalizer to standardize diverse regexes into a normal form for efficient training. Building on this, we propose SET2REGEX, a parameter-efficient (10M) base synthesizer equipped with a Hierarchical Set En-

[1]https://github.com/lorisdanto/automatark.

coder that explicitly enforces permutation invariance. Finally, we present a learning-based recursive framework comprising three specialized neural modules (ROUTER, PARTITIONER, and SEGMENTER) that learn to adaptively decompose synthesis tasks. This approach addresses the inherent complexity of optimal decomposition, which we prove to be NP-hard. Our empirical results demonstrate that RESYN significantly improves synthesis success rates on complex benchmarks, effectively bridging the gap between simplified research settings and real-world requirements.

Our contributions are summarized as follows:

- We quantitatively demonstrate the gap between existing benchmarks and real-world regexes, showing that they lack the structural complexity found in practice.
- We propose SET2REGEX, a parameter-efficient model (10M) that matches the performance of a large-scale baseline (300M) by leveraging a Hierarchical Set Encoder.
- We introduce RESYN, a recursive divide-and-conquer framework driven by learnable decomposition modules. Empirical results show that this framework significantly improves synthesis success rates on complex benchmarks like *RegExLib*.
- We prove that finding an optimal decomposition for regex synthesis is an NP-hard problem, highlighting the inherent computational intractability of the task and motivating the need for learned heuristics.

2 Related Work

Existing regex synthesis approaches can be broadly categorized into monolithic example-based methods and compositional divide-and-conquer strategies. We position our framework within the latter, explicitly addressing the scalability and structural limitations inherent in the former.

Example-Based and Neural Regex Synthesis. Traditional example-based synthesis treats the task monolithically, inferring a complete regex in a single step via exhaustive or evolutionary search [Bartoli *et al.*, 2014; Lee *et al.*, 2016]. However, these methods struggle to scale with the combinatorial search space of complex patterns [Oncina and García, 1992; Gold, 1978; Lang *et al.*, 1998]. Recent neural approaches frame synthesis as a sequence-to-sequence (Seq2Seq) problem, often leveraging frameworks like the Rational Speech Act (RSA) to improve accuracy [Vaduguru *et al.*, 2024]. Despite their inference speed, monolithic neural models face two critical hurdles. First, relying solely on sequential token generation hinders capturing deep, nested structures. Second, they typically ignore the permutation-invariant nature of input examples by flattening them into fixed sequences. This artificial linearity forces the model to learn spurious orderings instead of underlying shared structural patterns.

Divide-and-Conquer Synthesis. Alur et al. [Alur *et al.*, 2017] proposed synthesizing programs by hierarchically partitioning examples, where the induced decision structure mirrors the program's control flow. Farzan et al. [Farzan and Nicolet, 2021a; Farzan and Nicolet, 2021b] further demonstrated that decomposing synthesis tasks enables parallel solving, significantly reducing overall synthesis time. Similar principles appear in approaches that synthesize partial solutions over subsets of examples and compose them into a global solution [Barke *et al.*, 2020; Shrivastava *et al.*, 2021].

This paradigm has also been adapted for regex synthesis. Raza et al. [Raza *et al.*, 2015] decompose tasks using phrase-structure parsing of natural language descriptions, allowing sub-regexes to be synthesized independently. Chen et al. [Chen *et al.*, 2023] leverage LLMs to generate initial regex sketches, refining them by aligning components with substrings of positive examples. While these approaches highlight the benefits of decomposition, they rely on auxiliary supervision—such as natural language descriptions or externally generated sketches—limiting their applicability in purely example-driven settings.

Recent works have attempted decomposition using only input-output examples. To handle structurally diverse patterns that inherently require Union operators, earlier evolutionary approaches [Bartoli *et al.*, 2016] bypassed explicit structural decomposition by employing a data-level separate-and-conquer strategy, iteratively extracting subsets of examples and joining the resulting sub-expressions with top-level Unions. In contrast, recent decomposition-focused methods impose strong structural priors: FOREST [Ferreira *et al.*, 2021] utilizes heuristic-based common substring identification, while SPLITREGEX [Kim *et al.*, 2025] employs a neural model to predict segmentation points. However, a critical limitation of these modern approaches is that both rigidly assume that the top-level operator is always a Concatenation. This assumption severely hinders recursive decomposition. Because Concatenation is associative, repeatedly applying a Concatenation-only splitter results in a flat sequence of subproblems rather than a deep hierarchical structure. Consequently, these methods lack the flexibility to handle nested structures where Union operators appear at the top level or are interleaved with Concatenations.

Positioning of Our Work. RESYN advances compositional synthesis by introducing a generalized recursive framework. Unlike prior works that rely on fixed structural assumptions, we employ a learnable ROUTER that dynamically determines whether to decompose the problem via Concatenation (using a SEGMENTER) or Union (using a PARTITIONER). This flexibility allows for a principled, bottom-up composition that mirrors the complex syntactic structure of real-world regexes, achieved without requiring additional supervision.

3 Proposed Method

We present our recursive framework for example-based regex synthesis. We begin by introducing the definitions and notation used throughout the paper, followed by a formal problem formulation. We then describe our regex canonicalization pipeline, which standardizes diverse expressions into a unified representation. Next, we present SET2REGEX, a hierarchical encoder-decoder model that serves as the base synthesizer. Building upon this foundation, we introduce the RESYN framework, a recursive divide-and-conquer approach that decomposes complex synthesis problems into simpler sub-problems. Finally, we provide complexity analy-

sis demonstrating that finding optimal decompositions is NP-hard, which motivates our learning-based approach.

Definitions and Notation. Throughout the paper, we use standard notation for alphabets, strings, and regular expressions. An alphabet Σ is a finite set of symbols, and Σ^* denotes the set of all finite strings over Σ. A regular expression (regex) R describes a language $L(R) \subseteq \Sigma^*$. The formal, recursive definition of regex syntax and semantics, including all operators and character classes, is provided in Appendix A. We restrict our attention to regular constructs and exclude non-regular features such as backreferences.

We formally define the regex synthesis problem as follows: Given a set of positive examples $P \subseteq \Sigma^*$ and negative examples $N \subseteq \Sigma^*$, the goal is to find a concise regex R such that $P \subseteq L(R)$ and $N \cap L(R) = \emptyset$. While a trivial solution exists by simply enumerating all positive examples with union ($w_1|w_2|\cdots|w_n$ for $P = \{w_1, \ldots, w_n\}$), such a regex lacks generalization and is impractically verbose. Therefore, we seek a regex that is both correct and concise, capturing the underlying pattern rather than merely memorizing examples [Valizadeh *et al.*, 2024].

Three-Stage End-to-End Neural Recursive Synthesis Framework: ReSyn. We propose ReSyn, a comprehensive framework designed to overcome the structural complexity of real-world regexes. It is composed of three synergistic stages: data canonicalization, a synthesizer-agnostic recursive algorithm, and a specialized base model.

First, we address the issue of data quality. Real-world regex data exhibits significant syntactic diversity even when expressing identical languages, as different authors may write the same pattern in structurally different ways [Davis *et al.*, 2019]. Such unnecessary syntactic variation introduces noise that hinders model learning and reduces data efficiency. To address this challenge, we introduce a regex canonicalization pipeline that transforms diverse expressions into a standardized form. Full details of the pipeline are provided in Appendix B.

While our framework is compatible with any regex synthesizer, we propose Set2Regex, a parameter-efficient model designed to serve as a robust base synthesizer by capturing the permutation invariance of examples. As illustrated in Figure 1, the model utilizes a Hierarchical Set Encoder that processes examples in two stages. First, a character-level Transformer and Pooling by Multihead Attention (PMA) [Lee *et al.*, 2019] compress each string into a fixed-size embedding h_i. These embeddings are processed by a string-level Transformer, which replaces traditional positional encodings with type embeddings indicating whether the string is a positive or negative example. The resulting contextualized embeddings $\{h'_i\}$ are subsequently aggregated into a global context vector c. The decoder generates the regex using a two-stage attention mechanism, attending first to the global context c and then to the contextualized string embeddings $\{h'_i\}$.

Building on the canonicalized data and the base synthesizer, we introduce the core of our framework: the recursive decomposition algorithm (see Appendix C for the formal procedure and Figure A1 for the workflow). Unlike prior works that rely on fixed heuristics, ReSyn adaptively decomposes the problem using three specialized learnable modules: Router, Partitioner, and Segmenter. This modular design allows ReSyn to function as a wrapper that empowers any base synthesizer (including Set2Regex) to handle complex, nested patterns.

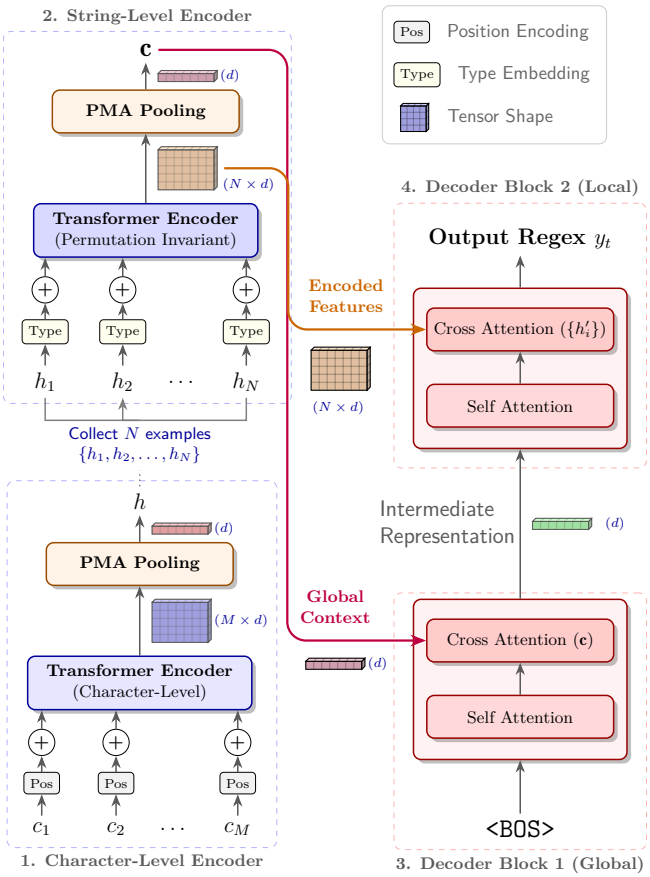

Figure 1: The architecture of Set2Regex. The Hierarchical Set Encoder aggregates character features into string embeddings (h_i) and subsequently into a global context (c) via PMA to ensure permutation invariance. The decoder employs a dual-attention mechanism, attending first to c for global structure and then to $\{h'_i\}$ for local details.

We define two complementary decomposition strategies utilized by the framework:

- **Segmentation:** This strategy divides each string in the positive example set P into k segments based on shared logical positions (Concatenation). Each string $w \in P$ is split into segments $s_1, \ldots, s_k$, transforming the original set P into an ordered sequence of k sets $(P_1, \ldots, P_k)$.
- **Partitioning:** This strategy addresses structural diversity by grouping similar strings together (Union). The set P is partitioned into m disjoint subsets $\{P_1, \ldots, P_m\}$, where each subset is treated independently.

Figure A2 (in Appendix) illustrates this recursive process. By synthesizing partial regexes at the leaves—using the chosen base synthesizer—and composing them bottom-up, the system constructs the final solution in a principled manner.

Specific architectural details for the learnable decomposition modules are provided in Appendix E.

Theoretical Hardness of Optimal Decomposition of Examples. To justify the use of neural architectures for regex synthesis, we establish the computational intractability of finding a concise regex. We define the *expression cost* $c_E(R)$ as the number of symbols (excluding operators) in regex R, and the *language expression cost* $c_E(S)$ as the minimum $c_E(R)$ such that $L(R) \supseteq S$.

We first relate the expression cost to the string alignment problem. An alignment of a set S into m tuples is a sequence $t_1 \dots t_m$ where each t_j corresponds to a single character $\sigma \in \Sigma$ or λ. The minimum alignment cost is denoted as $c(S)$.

Lemma 1. *For any finite language S, the language expression cost $c_E(S)$ is equal to the optimal alignment cost $c(S)$.*

Lemma 2. *The optimal alignment problem is NP-complete.*

Proof (Sketch). We reduce the *Shortest Common Supersequence* (SCS) problem to the optimal alignment problem. Given strings $\{w_1, \dots, w_n\}$, a common supersequence s of length m directly corresponds to an alignment of cost m. Since SCS is NP-complete [Räihä and Ukkonen, 1981], determining if $c(S) \leq r$ is also NP-complete. □

Based on Lemma 1 and 2, we obtain the following result.

Theorem 1. *The Concise Regex Problem, which decides whether $c_E(S) \leq r$ for a given set S and integer r, is NP-complete.*

Note that the technical details, including the proof of Theorem 1 are provided in Appendix F.

The NP-completeness implies that deterministic algorithms for optimal regex decomposition suffer from exponential time complexity as the number or length of examples increases. In practice, finding the global optimum through combinatorial search is often intractable.

This motivates the transition from an algorithmic approach to a learning-based approach. The proposed approach, RESYN, can learn to approximate the optimal alignment and decomposition patterns. By leveraging the generalizable representations of strings, neural architectures can efficiently navigate the vast search space of potential sub-regexes, providing near-optimal solutions in polynomial time where traditional symbolic methods fail to scale.

4 Experiments

4.1 Experimental Setup

Data Collection and Preprocessing. To train our neural models, we constructed a large-scale dataset of regular expressions by aggregating multiple sources: *Polyglot* [Davis *et al.*, 2019], *Python* [Chapman and Stolee, 2016], and the *StructuredRegex* [Ye *et al.*, 2020] dataset used in PRAX [Vaduguru *et al.*, 2024]. Among these, *Polyglot* and *Python* are real-world regexes collected from open-source repositories, while *StructuredRegex* is a synthetically generated dataset. For evaluation, we employed three benchmarks: *RegExLib* and *Snort*, which contain real-world regexes, and the test split of *StructuredRegex*, which is synthetic.

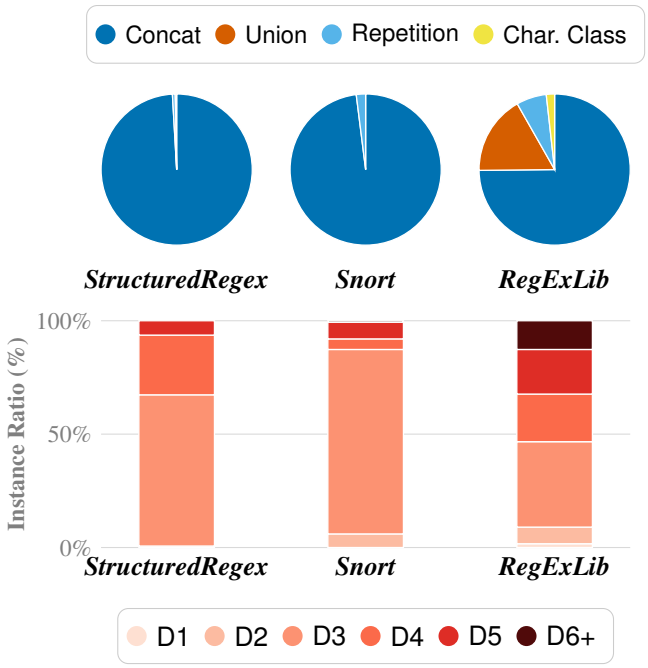

Figure 2: Structural Complexity Comparison across Benchmarks. The top row displays the distribution of Top-level Operators; while *StructuredRegex* and *Snort* are dominated by Concatenations, *RegExLib* exhibits a diverse structural composition with significant use of Union operators. The bottom row illustrates the AST Depth distribution using a sequential color gradient (darker indicates deeper nesting). Notably, *RegExLib* contains a significant proportion of high-complexity instances (Depth ≥ 5), whereas the other benchmarks are concentrated in shallower regions.

The detailed process of dataset construction, including canonicalization, structural filtering, sub-regex extraction, example generation (positive/negative), and substring expansion, is described in Appendix G. We highlight the structural characteristics of our evaluation benchmarks in Figure 2. Our evaluation suite consists of three distinct datasets: *StructuredRegex* (334 instances), *Snort* (352 instances), and *RegExLib* (1,752 instances). As illustrated in Figure 2, unlike the other baselines which are dominated by shallow Concatenations, *RegExLib* exhibits a diverse range of operators and significantly deeper AST structures, posing a greater challenge for synthesis.

Baselines. Since traditional enumerative synthesizers scale poorly to the complex, nested regular expressions targeted in this work, we focus our comparison exclusively on neural synthesis approaches. We compare our framework against two categories of baselines: neural synthesizer models and decomposition-based approaches.

We evaluate against the following state-of-the-art neural synthesis models:

- PRAX: A ByT5-based Seq2Seq model [Vaduguru *et al.*, 2024]. For a fair comparison, we fine-tuned `google/byt5-small` on our dataset for 10 epochs using BF16 precision and an effective batch size of 64 via gradient accumulation.

- `gpt-oss-120b` & `GPT-5`: Large-scale reasoning models evaluated via identical zero-shot prompting (Appendix H). Due to its immense capacity, `GPT-5` is benchmarked separately against our beam search model to highlight our framework's efficiency rather than in the main comparison.

We compare our recursive decomposition strategy against approaches that employ problem splitting to validate the effectiveness of the strategy. For fair comparison, we adopted only their decomposition strategies and combined them with neural synthesizer backend.

- SPLITREGEX [Kim *et al.*, 2025]: A neural strategy that segments positive strings. Since SPLITREGEX is functionally identical to our SEGMENTER, we only include it as a baseline in the ablation study.
- FOREST [Ferreira *et al.*, 2021]: A heuristic approach that identifies common substrings as separators to split the synthesis task.
- Oracle: Oracle variant of RESYN, performing recursive decomposition guided by the ground truth regex structure. It follows the same inference pipeline and termination logic as RESYN but utilizes ground truth labels for routing and decomposition decisions to establish an upper bound on performance.

Implementation Details. Our framework is highly parameter-efficient, totaling only 29.6M parameters (SET2REGEX: 10M, ROUTER: 4.6M, PARTITIONER: 7.5M, SEGMENTER: 7.5M). We utilized greedy decoding by default for inference. Detailed hyperparameters, architecture sizes, and training data coverage are provided in Appendix I.

Evaluation Metrics

Evaluating synthesized regular expressions in a PBE setting presents unique challenges. A trivial solution that simply enumerates all positive examples using the Union operator satisfies the input constraints but fails to generalize and merely memorizes the data without capturing the underlying structure. To comprehensively assess the quality of the synthesized regexes, we employ three complementary metrics that measure structural conciseness, strict functional equivalence, and approximate functional similarity.

We define the **Synthesis Success Rate** as the percentage of instances where the synthesized regex R_{pred} correctly classifies all training examples. Additionally, to penalize trivial enumerations, we measure the **Conciseness Ratio**, defined as $|R_{pred}|/|R_{gt}|$, where R_{gt} denotes the ground truth regex. Ratios closer to 1.0 indicate successful pattern generalization rather than mere data memorization. To evaluate strict generalization, we use **Semantic Accuracy**, defined as the percentage of cases where the synthesized regex correctly classifies *all* examples in the held-out set. This is the strictest criterion, requiring the synthesized regex to be functionally equivalent to the ground truth within the scope of the test data. However, because Semantic Accuracy can be overly punitive due to the under-specification problem (e.g., distinguishing `[a-z]*` from `[a-z]+` without empty string examples), we also measure the **Matthews Correlation Coefficient (MCC)** on the held-out set:

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN} - \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}} \tag{1}$$

This provides a balanced correlation metric ranging from -1 to +1. Finally, regarding **Failure Handling**, failed syntheses are recorded as failures for the Success Rate. For secondary metrics (Conciseness and MCC), failed instances are treated as a trivial union of positive examples, naturally penalizing them with excessive length and poor generalization scores.

4.2 Experimental Results and Analysis

We evaluate the proposed methods by addressing the following four research questions (RQs):

- **RQ1:** Can the RESYN framework universally improve the performance of neural synthesizers?
- **RQ2:** Is recursive decomposition more effective than non-recursive (top-level only) splitting for complex real-world regexes?
- **RQ3:** Does the hierarchical architecture of SET2REGEX achieve parameter efficiency while maintaining performance comparable to existing neural baselines?
- **RQ4:** Can our specialized framework compete with large-scale general-purpose reasoning models?

Table 2 summarizes the quantitative results of our experiments across three benchmarks of increasing structural complexity: *StructuredRegex*, *Snort*, and *RegExLib*. To rigorously evaluate the impact of decomposition, we employ two distinct base synthesizers: the existing PRAX model and our proposed SET2REGEX. For each base synthesizer, we compare our RESYN framework against the decomposition-based baseline, FOREST, to isolate the efficacy of our recursive approach.

Overall, the results demonstrate that applying RESYN consistently improves performance across base models, outperforming the FOREST baseline in most metrics. Notably, the SET2REGEX + RESYN configuration achieves state-of-the-art results, demonstrating performance comparable to or even surpassing the massive general-purpose reasoning model on the most complex benchmark across the majority of evaluation metrics, despite having significantly fewer parameters.

RQ1: Effectiveness of ReSyn Framework

RESYN significantly enhances neural synthesizers across all benchmarks (Table 2). For PRAX, it boosts the *RegExLib* success rate from 42.24% to 67.29%. The impact is greater for our SET2REGEX model, where the SET2REGEX + RESYN configuration achieves a 68.26% success rate (+29.33% absolute increase) and improves Semantic Accuracy to 41.61%. These gains are most pronounced in the complex real-world dataset *RegExLib*, where RESYN's recursive decomposition effectively mitigates structural complexity. Furthermore, the lower Conciseness Ratio indicates that RESYN encourages compact, generalized patterns rather than trivial data memorization.

Method	*StructuredRegex*				*Snort*				*RegExLib*			
	Succ	Conc	Acc	MCC	Succ	Conc	Acc	MCC	Succ	Conc	Acc	MCC
PRAX	85.03	2.16	76.05	83.47	67.05	6.21	56.53	64.33	42.24	5.27	33.68	39.71
PRAX + Forest	80.84	2.35	67.96	78.25	77.84	5.01	**63.07**	73.01	49.94	4.76	36.76	46.24
PRAX + RESYN	96.71	**1.35**	84.43	94.29	**80.68**	**4.64**	61.93	**74.65**	67.29	4.00	40.81	58.75
SET2REGEX	90.12	1.75	82.34	88.61	55.40	7.92	44.03	51.29	38.93	5.72	30.99	36.08
SET2REGEX + Forest	85.03	2.02	73.95	82.72	76.70	5.31	61.93	71.41	49.03	4.99	35.62	44.92
SET2REGEX + RESYN	**97.60**	1.38	**85.93**	**95.23**	79.55	4.91	58.24	72.41	**68.26**	**3.88**	**41.61**	58.97
`gpt-oss-120b`	79.94	2.70	49.10	71.81	43.18	8.33	29.26	39.36	67.29	4.03	40.75	**59.06**
PRAX + Oracle	99.10	1.14	87.43	96.78	86.36	3.62	65.91	79.64	80.59	3.29	41.72	65.40
SET2REGEX + Oracle	99.10	1.21	88.62	96.99	86.36	3.57	63.92	78.55	82.53	2.85	43.15	66.51

Table 2: Comprehensive comparison of synthesis methods across benchmarks. Succ: Synthesis Success Rate (%), Conc: Conciseness Ratio (lower is better), Acc: Semantic Accuracy (%), MCC: Matthews Correlation Coefficient (×100). Best results in bold, second best underlined (excluding Oracle). Highlighted rows indicate our proposed adaptive decomposition (RESYN) and the Oracle upper bound.

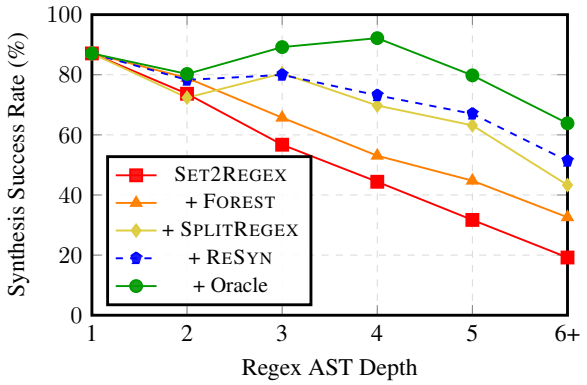

Figure 3: Performance comparison across regex AST depth. Non-recursive methods (FOREST, SPLITREGEX) show sharp degradation with increasing depth, while Recursive maintains robust performance. The gap widens beyond depth 4, highlighting the necessity of recursive decomposition for complex real-world regexes.

RQ2: Importance of Recursive Decomposition

We compared RESYN against single-level decomposition baselines (SPLITREGEX, FOREST). As shown in Figure 3, non-recursive methods exhibit sharp performance degradation as regex AST depth increases. While shallow splitting suffices for simple patterns (Depth ≤ 3), it fails to address the combinatorial complexity of nested structures. In contrast, RESYN maintains robust performance even at Depth 5 and 6+, validating the necessity of a recursive strategy.

RQ3: Parameter Efficiency of Hierarchical Architecture

SET2REGEX (10M) is 30× smaller than the PRAX baseline (300M) but achieves comparable or superior performance. This efficiency stems from our Hierarchical Set Encoder, which eliminates the need to learn spurious example orderings, a common overhead in traditional Seq2Seq models. Notably, SET2REGEX outperforms PRAX on the synthetic benchmark (90.12% vs. 85.03%) and matches it on *RegExLib*, proving that architectural alignment with set-theoretic inputs can decouple model capacity from synthesis accuracy.

RQ4: Comparison with Advanced Language Models

Compared to `gpt-oss-120b`, our 29.6M-parameter framework achieves higher Synthesis Success Rates and Semantic Accuracy across all benchmarks despite being $4,000\times$ smaller and avoiding expensive reasoning tokens. We further evaluated RESYN (with $k = 500$ beam search) against `GPT-5`. As shown in Table 3, RESYN outperforms `GPT-5` on *StructuredRegex* and *Snort*, and notably achieves higher Semantic Accuracy on *RegExLib*. This indicates that our recursive inductive bias captures ground-truth structures more effectively than general-purpose large-scale reasoning, ensuring superior generalization with a fraction of the computational footprint.

4.3 Ablation Study

We analyze the contribution of each component by evaluating five configurations based on SET2REGEX: (1) RESYN: the full framework with dynamic recursive routing; (2) w/o ROUTER: a fixed heuristic that rigidly alternates between Segmentation and Partitioning; (3) SEGMENTER only: single-level concatenation splitting (equivalent to SPLITREGEX); (4) PARTITIONER only: single-level union splitting; and (5) SET2REGEX: the base model without decomposition.

Impact of Decomposition Strategies. Table 4 summarizes the results on the *RegExLib* benchmark. Comparing the non-recursive baselines, SEGMENTER only (65.30%) significantly outperforms PARTITIONER only (40.87%), confirming that Concatenation is the dominant top-level operator in regexes. However, the full RESYN framework achieves a higher MCC (58.97) than SEGMENTER only (58.81). This marginal yet crucial gain (0.16) demonstrates that while segmentation handles most structures, the PARTITIONER is essential for capturing the semantic branching of Union operators, which a Concatenation-only approach fails to represent.

To understand this marginal 0.16 MCC gain despite RESYN's structural superiority, consider the following target regex:

- **Target:** `\d{1,2}|\d{1,2}\.\d{1,3}|\x7f`
- RESYN: `[2-9]\d?|[1-8]\d\.\d{1,3}|\x7f`
- SEGMENTER **only:** `\d?\d?[\.\x7f]?(\d{3})?`

Method	*StructuredRegex*				*Snort*				*RegExLib*			
	Succ	**Conc**	**Acc**	**MCC**	**Succ**	**Conc**	**Acc**	**MCC**	**Succ**	**Conc**	**Acc**	**MCC**
Large-scale Reasoning Model (N/A Params)												
GPT-5	96.71	1.43	60.78	87.59	85.23	3.70	54.83	73.55	**90.07**	**2.12**	48.86	**75.89**
Specialized Neural Synthesis (SET2REGEX + RESYN, 29.6M Params)												
Beam Search ($k = 500$)	**100.00**	**1.02**	**89.52**	**97.79**	**90.62**	**3.28**	**63.64**	**82.10**	85.33	2.58	**50.29**	73.65

Table 3: Comparison between `GPT-5` and RESYN using beam search decoding ($k = 500$). While `GPT-5`'s exact parameter count is undisclosed, our highly parameter-efficient RESYN framework significantly narrows the performance gap. Notably, beam search enables RESYN to achieve competitive or even superior performance with a fraction of the computational footprint. Succ: Synthesis Success Rate (%), Conc: Conciseness Ratio (lower is better), Acc: Semantic Accuracy (%), MCC: Matthews Correlation Coefficient ($\times 100$). Best results in bold.

Method	Configuration		RegExLib	
	Strategy	**Recursion**	**Succ**	**MCC**
RESYN	Adaptive	Yes	<u>68.26</u>	**58.97**
w/o ROUTER	Fixed	Yes	**73.63**	49.34
SEGMENTER only	Seg. Only	No	65.30	<u>58.81</u>
PARTITIONER only	Part. Only	No	40.87	37.44
SET2REGEX	-	No	38.93	36.08

Table 4: Ablation study on the *RegExLib* benchmark quantifying the impact of recursive modules. Succ: Synthesis Success Rate (%), MCC: Matthews Correlation Coefficient ($\times 100$). Best results in bold, second best underlined.

While RESYN correctly deduces the structural branches, suffering a slight MCC penalty (81.65) due to minor digit-range overfitting, SEGMENTER only collapses the Union, overgeneralizing to a malformed regex yet paradoxically achieving a higher MCC (90.45). This occurs because edit-distance-based negatives lack structural edge cases. Since mitigating this with computationally prohibitive techniques like Rational Speech Act (RSA) is infeasible for massive datasets, the seemingly small MCC difference actually masks a significant leap in structural correctness.

Efficacy of the Learnable Router. A critical insight arises from comparing RESYN with the fixed heuristic w/o ROUTER. While the fixed strategy achieves the highest synthesis success rate (73.63%), its MCC drops drastically to 49.34 (-9.63 compared to RESYN). This discrepancy reveals that the fixed heuristic suffers from over-decomposition: it aggressively fragments the problem into trivial sub-problems, satisfying the training examples but destroying the underlying generalization structure. In contrast, the learned ROUTER in RESYN successfully avoids spurious splits, balancing solvability (Success Rate) with structural integrity (MCC).

4.4 Case Study and Limitations

We present a qualitative case study to highlight where RESYN's recursive decomposition excels, followed by a discussion on current limitations in evaluation metrics.

Union Structure within Concatenation

- **Ground Truth:** `(I{2,10}|V{2,10})[A-Z]{3,4}`
- RESYN: `(I{2,10}|V{2,10})[A-Z]{3,4}`
- **gpt-oss-120b:** `(V{2,}[A-Z]*V?|I{2,}[A-Z]*I?)`

Analysis: This case highlights the efficacy of the PARTITIONER. First, the framework decomposes the problem into a prefix and a suffix via Concatenation. While the suffix is uniform (`[A-Z]{3,4}`), the prefix contains distinct subgroups. RESYN successfully partitions the prefix into 'I-based' and 'V-based' clusters, synthesizing the correct Union logic. In contrast, the baseline fails to disentangle the groups, resulting in an overfitted and convoluted regex.

Limitation and Future Work

Current evaluation metrics like MCC can penalize structurally correct models because negative examples lack structural edge cases, allowing malformed regexes to score highly. Future work will explore semantic hard-negative mining to better evaluate and unlock RESYN's full potential, moving beyond simple edit-distance perturbations.

5 Conclusions

In this work, we bridged the gap between existing neural regex synthesizers and the complexity of real-world regular expressions. Our statistical analysis highlighted that previous benchmarks fail to reflect the intricate, nested structures found in practical applications. To overcome the limitations of sequential generation models on such data, we proposed RESYN, a novel framework that combines a robust base synthesizer with a recursive decomposition strategy.

Our contributions are threefold. First, we introduced a rigorous data processing pipeline, including regex canonicalization and structural de-duplication, ensuring high-quality training and fair evaluation. Second, we designed SET2REGEX, which utilizes a hierarchical encoder to model the set-based nature of PBE inputs, achieving state-of-the-art efficiency with $30\times$ fewer parameters than baselines. Third, we demonstrated that our recursive framework, driven by a learnable ROUTER, effectively solves the NP-hard decomposition problem. Empirically, while non-recursive methods struggle with the deep nesting of real-world regexes, our recursive approach adapts the problem size to the model's capability, yielding significant gains in challenging scenarios.

We believe our divide-and-conquer methodology offers a generalizable direction for program synthesis beyond regular expressions, particularly when target programs exhibit recursive structures.

Acknowledgements

Sang-Ki Ko and Seongmin Kim were supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. RS-2024-00456065). Yo-Sub Han and Su-Hyeon Kim were supported by the NRF grant (No. RS-2025-00562134) and the AI Graduate School Program (No. RS-2020-II201361) funded by the Korean government.

References

[Alur *et al.*, 2017] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS*, pages 319–336, 2017.

[Barke *et al.*, 2020] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[Bartoli *et al.*, 2014] A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio. Automatic synthesis of regular expressions from examples. *Computer*, 47(12):72–80, 2014.

[Bartoli *et al.*, 2016] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, 2016.

[Chapman and Stolee, 2016] Carl Chapman and Kathryn T Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 282–293, 2016.

[Chen *et al.*, 2023] Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Işıl Dillig. Data extraction via semantic regular expression synthesis. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):1848–1877, 2023.

[Davis *et al.*, 2019] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In *ESEC/FSE*, pages 443–454, 2019.

[Farzan and Nicolet, 2021a] Azadeh Farzan and Victor Nicolet. Counterexample-guided partial bounding for recursive function synthesis. In *CAV*, pages 832–855, 2021.

[Farzan and Nicolet, 2021b] Azadeh Farzan and Victor Nicolet. Phased synthesis of divide and conquer programs. In *PLDI*, pages 974–986, 2021.

[Ferreira *et al.*, 2021] Margarida Ferreira, Miguel Terra-Neves, Miguel Ventura, Inês Lynce, and Ruben Martins. Forest: an interactive multi-tree synthesizer for regular expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 152–169. Springer, 2021.

[Gold, 1978] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[Kim *et al.*, 2025] Seongmin Kim, Hyunjoon Cheon, Su-Hyeon Kim, Yo-Sub Han, and Sang-Ki Ko. Splitregex: Efficient regex synthesis by neural example splitting. *Journal of Automata, Languages and Combinatorics*, 30(1–3):157–179, 2025.

[Lang *et al.*, 1998] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *ICGI*, pages 1–12, 1998.

[Lee *et al.*, 2016] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *GPCE*, pages 70–80, 2016.

[Lee *et al.*, 2019] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pages 3744–3753. PMLR, 2019.

[Oncina and García, 1992] J. Oncina and P. García. Inferring regular languages in polynomial updated time. In *Pattern Recognition and Image Analysis*, pages 49–61. World Scientific, 1992.

[Räihä and Ukkonen, 1981] Kari-Jouko Räihä and Esko Ukkonen. The shortest common subsequence problem over binary alphabet is np-complete. *Theoretical Computer Science*, 16(2):187–198, 1981.

[Raza *et al.*, 2015] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI 2015*, 2015.

[Shrivastava *et al.*, 2021] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis. In *NeurIPS*, pages 6102–6114, 2021.

[Vaduguru *et al.*, 2024] Saujas Vaduguru, Daniel Fried, and Yewen Pu. Generating pragmatic examples to train neural program synthesizers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[Valizadeh *et al.*, 2024] Mojtaba Valizadeh, Philip John Gorinski, Ignacio Iacobacci, and Martin Berger. Correct and optimal: The regular expression inference challenge. In Kate Larson, editor, *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 6486–6494. International Joint Conferences on Artificial Intelligence Organization, 8 2024. Main Track.

[Ye *et al.*, 2020] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Benchmarking multimodal regex synthesis with complex structures. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational*

Linguistics, pages 6081–6094, Online, July 2020. Association for Computational Linguistics.